### Why Microservices

Traditionally, services were designed as monoliths with all endpoints supported by a single service. A consequence of this approach is that unrelated features of the same product end up in the same server. For example, the service endpoints supporting your product’s reporting needs are not necessarily related to its login or billing endpoints. Yet, they all reside in the same server.

An obvious problem with this architecture is that every minor change in a small part of the code requires deploying the entire code base again. As the product gets bigger, development and deployment become cumbersome with this approach. To make changes and deploy them, developers have to coordinate with every team that works on the server code. This approach is obviously not scalable or agile enough for fast development cycles.

This is where microservices are relevant. In a microservice-based architecture, rather than keeping all the modules together in the same monolithic service, you design separate smaller services for different modules. For example, you can have a small service take care of your billing requirements and another independent service for login. When one module requires information from another, it can just make an API call to the corresponding microservice.

With this decoupling, developers can focus on their particular microservice and deploy it as and when required, rather than waiting for changes from a different module to be finished. Teams can have complete ownership of their entire code base, plus a smaller code base is much easier to maintain. If something goes wrong with a particular microservice, it only brings down that microservice; the rest of the independent microservices will be up and running.

Another advantage is the ability to scale microservices separately. Not all endpoints are used the same amount. For example, the login service is probably used significantly more than, say, billing services are. If we follow a monolith approach, we have to scale up the entire service, thus wasting resources. With a microservice-based approach, we just have to scale up the required microservice. In the above example, we could have a lot more instances of the logging microservice than the billing microservice.

When we decided to move to a microservice architecture at Clover, different designs came up for discussion. One approach we tried was to use Spring Boot along with Spring Cloud. This article discusses the basics of creating a microservice using Spring Boot and Spring Cloud. Spring Boot helps you to create a Spring application with minimal effort, while Spring Cloud provides you with a set of tools that makes communication between different microservices easier.

### ****Spring Boot****

Spring Boot is an efficient framework for creating a Spring-based application. Anyone with basic Java programming skills can quickly create and run a Spring Boot microservice. The learning curve is small, yet it can be used to create robust production-grade applications. Spring Boot and Spring Cloud provide many built-in tools and frameworks that make developing a cloud-based microservice rather easy. Most features can be enabled by just using few annotations, thus making development quite agile.

#### *Creating a simple Spring Boot application*

The easiest way to create a Spring Boot application is to use the [Spring Initializr](https://start.spring.io/). Spring provides you with an easy-to-use interface to generate a basic Spring Boot project with all required dependencies.

You can choose either Maven or Gradle as your build tool. You can select either Java, Kotlin, or Groovy as the programming language, plus your desired Spring Boot version. You can also specify your group and artifact IDs and the dependencies you need.

Spring Initializr will generate a basic Maven/Gradle project that you can import into Eclipse or any other IDE to use as a starting point for your project.

The generated project will have a class (usually named Application) with @SpringBootApplication annotation. This is the starting point of the Spring Boot application. It looks something like this:

The rest of the development is exactly like how you would develop a Spring application. You define your controller layer and use annotations to map requests to methods. The Spring Boot application will pick up annotations and route your requests to the appropriate methods.

Sample request handler:

Use Maven/Gradle builds to generate JARs/WARs to deploy.

#### Deploying your Spring Boot application

* **Embedded servers:**A Spring Boot application can be deployed with an embedded server. You can generate a JAR file and run it like any other JAR file. This avoids having to run and maintain a standalone server instance. Further making this easier, Spring Boot provides support for embedded Jetty, Undertow, and Tomcat, which can be configured in pom.xml (or Gradle), like this:

<dependency>   
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-jetty</artifactId>  
</dependency>

* **Standalone servers:** You can also deploy a Spring Boot application in a standalone server (e.g., standalone Tomcat). For this, you can generate a WAR file for your Spring Boot application (instead of a JAR) and deploy it like a normal WAR file. This will be covered in a separate post.

### ****Spring Cloud****

One of the key challenges in deploying a microservice is handling smooth communication between different microservices. One might require load balancers, some sort of central registry that keeps track of which microservices are up or down, and error handling in case of broken connections, and so on. Fortunately, Spring Cloud provides tools and frameworks to accomplish this easily. Some of the basic Spring Cloud features we tested out are service discovery with Eureka, load balancing with Ribbon, and circuit-breaking with Hystrix.

#### Eureka

One advantage of deploying a microservice is the ability to horizontally scale individual services as and when required. When new instances of a service are spun up dynamically, the other services that use it need to know about the additional availability. Obviously hardcoding addresses of instances of services is not scalable. This is where a service discovery tool comes into play. Service discovery tools allow individual microservices to register themselves with a registry when they are deployed and running. When a client requires a particular service, it can obtain a list of instances of the service from the discovery tool and then query an instance of the service.

Eureka is one such easy to use service discovery tool. The first step in using Eureka is starting an Eureka server. This server acts as the registry to which other services register themselves. When one microservice needs to communicate with another, it can query Eureka to find the active instance of that microservice.

The Eureka server can be another Spring Boot application. Again, you can use [Spring Initializr](https://start.spring.io/) to create this application. Remember to choose Eureka as a dependency in Spring Initializr when you generate your Eureka server application.

After generating the application, edit the application to add annotation @EnableEurekaServer to your Application class (seen in the first gist).

You configure your server port in application.yml. For a local set up, it looks like:

server:   
 port: 8086

eureka:  
 instance:  
 hostname: localhost  
client:  
 registerWithEureka: false  
 fetchRegistry: false  
 serviceUrl:  
 defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

From the browser, go to http://localhost:port/ to see your Eureka dashboard.

Once you have the Eureka server running, the next step is adding Eureka changes to your microservices so that they register with your Eureka server when deployed. To do this, first add the Eureka dependency to pom.xml:

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
 <version>2.0.0.RELEASE</version>  
</dependency>

Then make changes in application.yml to add Eureka server configurations. An example for local set up:

eureka:  
 client:  
 registerWithEureka: true  
 fetchRegistry: true  
 serviceUrl:  
 defaultZone: <http://localhost:8086/eureka>  
 instance:  
 hostname: localhost

At the end, add the annotation @EnableDiscoveryClient to your Applicationclass.

Start the service and check the Eureka dashboard. Your service should appear in the list. Optionally, you can start more instances of the service to check that they show up on the dashboard.

#### Ribbon

Once we have Eureka up and running, when your service wants to query another service, it should get a list of instances of the second service from Eureka server and call one of them.

This process is greatly simplified by the use of another tool called Ribbon. Ribbon is a load balancer, which can be configured to automatically obtain a list of instances of a service from Eureka and query the service while balancing the load.

All of your queries to another microservice will be routed through Ribbon, which talks with Eureka to find the actual address of the microservice instance that you need to query. Ribbon can be enabled through application.yml:

ribbon:  
 https:  
 client:  
 enabled: true

Sample code for calling another service using Ribbon and Eureka is given below. In this example, our function calls a second service which is registered with the name DEMOB on Eureka. (Note the serviceUrl: it uses this name and not some address; Ribbon and Eureka take care of resolving the address.)

Also note that we create a RestTemplate bean with @LoadBalanced and @Beanannotations (autowired to the restTemplate protected object in line 1). This is essential to make RestTemplate use Ribbon for load balancing.

#### Hystrix

After setting up your microservices, they obviously need to communicate with each other over the network. Even after your best efforts, API calls to a microservice could fail due to a variety of reasons.

In such situations, you can use Hystrix to provide some level of fault tolerance. Hystrix is a circuit-breaker. Hystrix lets you define a fallback method that gets invoked if your network calls to another microservice fails. It reverts back to normal behavior once the service is available again.

To enable Hystrix, add the following dependencies to pom.xml. The first dependency lets you add Hystrix, while the second dependency lets you enable a Hystrix dashboard, which lets you monitor the health of the service. The actuator dependency is required to collect metrics for Hystrix.

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>  
 <version>2.0.0.RELEASE</version>  
</dependency>  
   
<dependency>  
 <groupId>org.springframework.cloud</groupId>\  
 <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>  
 <version>2.0.0.RELEASE</version>  
</dependency>  
   
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

To use the Hystrix dashboard, add the following to application.yml:

management:  
 endpoints:  
 web:  
 exposure:  
 include: hystrix.stream

After this, add annotations @EnableHystrixDashboard and @EnableCircuitBreaker to the Application class.

The following code shows how to write a Hystrix wrapper around restTemplate. Inside the callGetService function, if the restTemplate call fails, the control moves to the fallback method mentioned in the @HystrixCommand annotation. (This is the same code used above to illustrate the use of Ribbon.)

### Conclusion

In short, creating a microservice using Spring Boot is rather easy. Most of the work of finding and adding dependencies is taken care of by Spring Initializr. If you are familiar with Spring development, moving to Spring Boot is straightforward. You can enable most of the features using just annotations instead of complex configurations. The time required for learning Spring Boot and building a microservice with it is minimal. Microservices provide an easy way to move away from large monolithic services toward easily maintainable small services, and Spring Boot provides an effective way to build and maintain them.

**Spring Boot Microservices: Building a Microservices Application Using Spring Boot**

Now that we've learned how to set up and run a Spring Boot app using Eclipse IDE and CLI, we'll see how to build a microservices application using Spring Boot.

Join the DZone community and get the full member experience.

[**JOIN FOR FREE**](https://dzone.com/articles/spring-boot-microservices-building-microservices-a)

In a previous pst, we learned how to setup and run [Spring Boot using Eclipse IDE and CLI](https://www.edureka.co/blog/spring-boot-setup-helloworld-microservices-example/). Now in this Spring Boot Microservices post, let me show how we can create Microservices Application for Top Sports Brands using Spring Boot and Netflix Eureka Server in detail. Before creating the application, let me tell you what are the challenges with Microservices Architecture.

Spring Boot enables building production-ready applications quickly and provides non-functional features:

* Embedded servers which are easy to deploy with the containers.
* It helps in monitoring the multiples components.
* It helps in configuring the components externally.

So, let us see the challenges with [microservices architecture](https://dzone.com/articles/microservices-vs-soa-whats-the-difference).

**Challenges With Microservice Architecture**

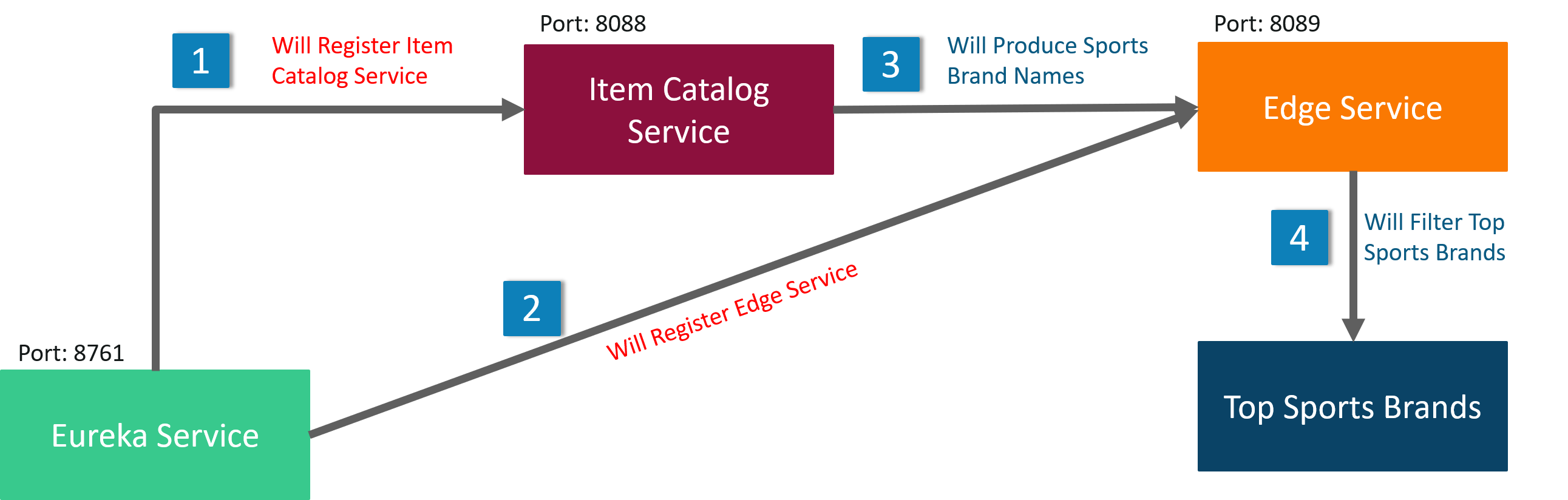
While developing a number of smaller microservices might look easy, there is a number of inherent complexities that are associated with microservices architectures. Let's look at some of the challenges:

* **Automating the Components**: It becomes difficult to automate everything because there are a number of smaller components instead of a monolith, i.e. builds, deployment, monitoring, etc.
* **Perceptibility**: There is a number of small components to deploy and maintain which sometimes becomes difficult to monitor and identify problems. It requires great perceptibility around all the components.
* **Configuration Management**: There is a great need to maintain the configurations for the components across the various environments.
* **Debugging**: It becomes difficult to probe each and every service for an error. Centralized Logging and Dashboards are essential to make it easy to debug problems.
* **Consistency**: You cannot have a wide range of tools solving the same problem. While it is important to foster innovation, it is also important to have some decentralized governance around the languages, platforms, technology and tools used for implementing/deploying/monitoring microservices.

**Building Architecture for Top Sports Brands With Spring Boot**

In this Spring Boot microservices example, we will be creating Top Sports Brands' application, which will have three services:

1. **Eureka Service**- This service will register every microservice and then the client microservice will look up the Eureka server to get a dependent microservice to get the job done. This Eureka Server is owned by Netflix and in this, Spring Cloud offers a declarative way to register and invoke services by Java annotation.
2. **Item Catalog Service -**This service will generate the list of sports brands which are popular in the market.
3. **Edge Service -**It is similar to the standalone Item service created in Bootiful Development with Spring Boot and Angular. However, it will have fallback capabilities which prevent the client from receiving an HTTP error when the service is not available

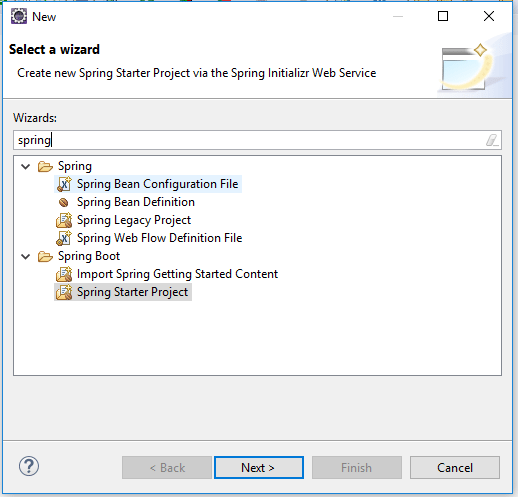


Let us see which of the following tools required to create this Spring Boot microservices example application.

If you facing any difficulty in installing and running the above tools, please refer to this blog.

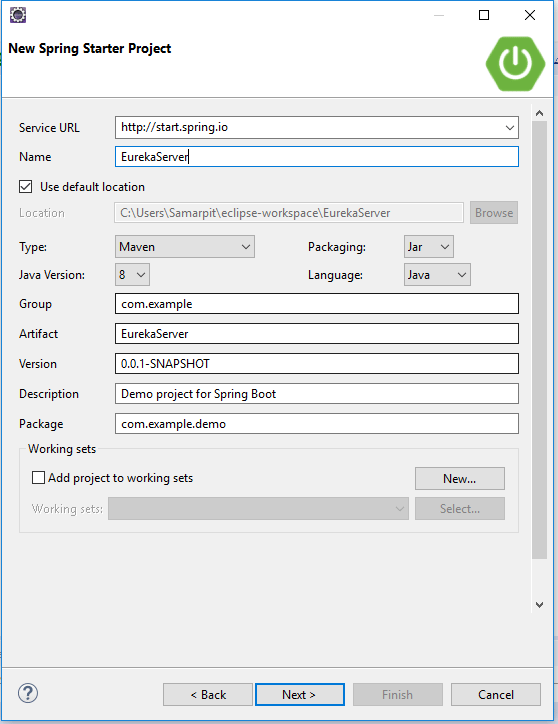
**Creating a Eureka Service**

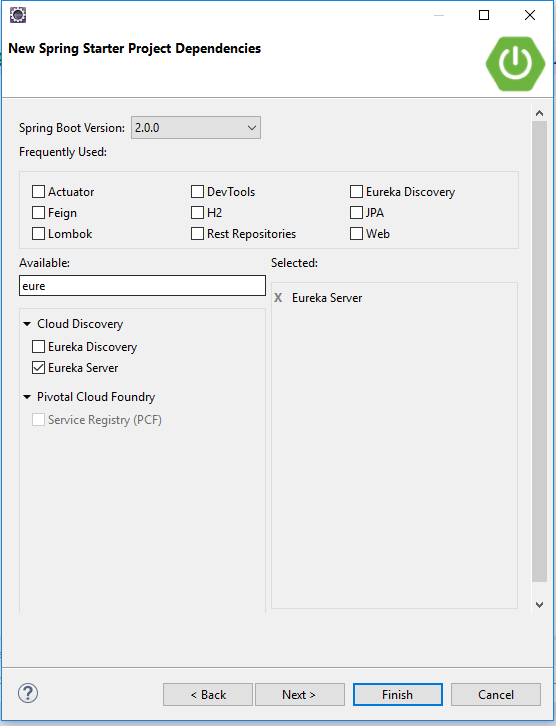
To begin with, create a EurekaServer Spring Starter Project in Eclipse IDE. Click on Spring Starter Project and click on Next.



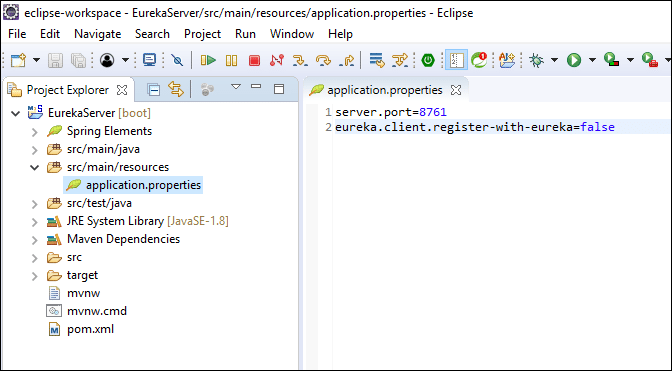
Name your Spring Starter Project as **EurekaServer**and other Information will be filled automatically.

**Note:**Make sure your Internet is connected otherwise it will show an error.





Now, modify EurekaServer/src/main/resources/application.properties file to add a port number and disable registration.

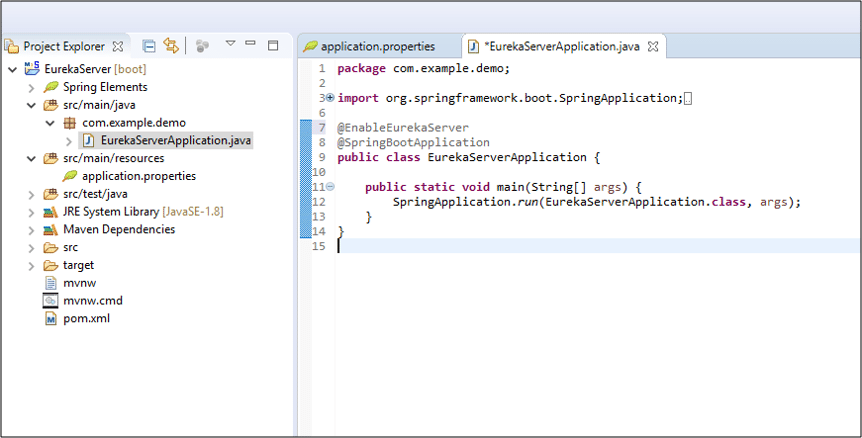


Open EurekaServer/src/main/java/com/example/EurekaServiceApplication.java and add @EnableEurekaServerabove @SpringBootApplication.

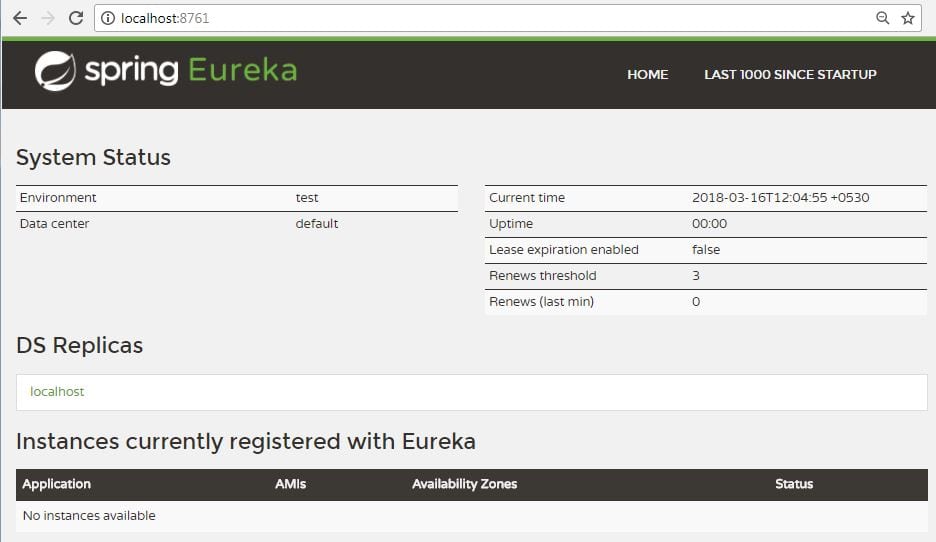
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@EnableEurekaServer @SpringBootApplication

This annotation will configure a registry that will allow other applications to communicate.

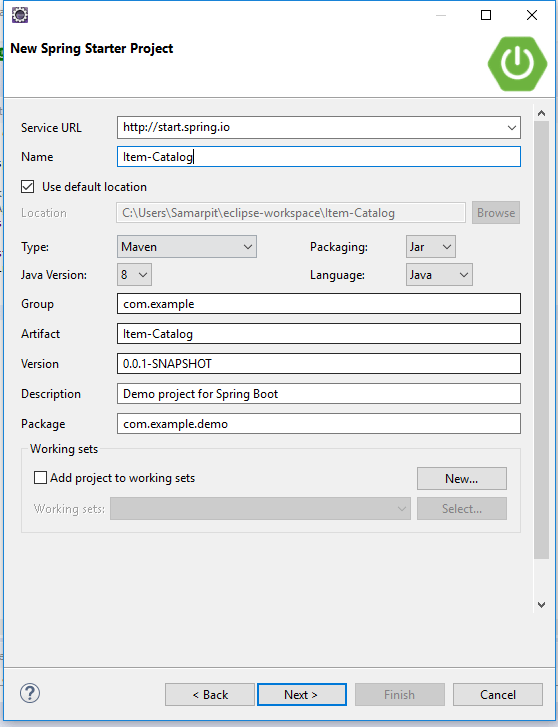
To start the Application: Right Click on theProject -> Run As -> Click on " Spring Boot App "

http://localhost:8761

Now open http://localhost:8761. Here Spring Eureka Server will open and will show no service will be running.

**Spring Boot Microservices: Creating an Item Catalog Service**

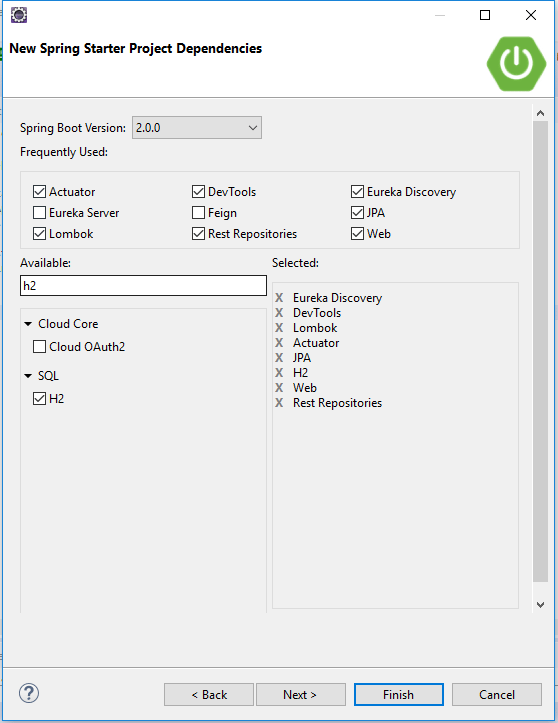
Again create a new project. Use Item-catalog-service for the artifact name and click on **Next**.



Add the following dependencies:

* **Actuator**: features to help you monitor and manage your application
* **EurekaDiscovery**: for service registration
* **JPA**: to save/retrieve data
* **H2**: an in-memory database
* **RestRepositories**: to expose JPA repositories as REST endpoints
* **Web**: Spring MVC and embedded Tomcat
* **DevTools**: to auto-reload the application when files change
* **Lombok**: to reduce boilerplate code

Click on **Finish.**



Now, create an entity, to ItemCatalogServiceApplication.java . The code below assumes you're putting all classes in the same file.

If you're using an editor that doesn't auto-import classes, here's the list of imports needed at the top of ItemCatalogServiceApplication.java.

Add an application name in item-catalog-service/src/main/resources/application.properties file to display in the Eureka service, and set the port to 8088.

Now, Create the Cloud Properties file.

Click on**File -> New -> Other -> File**and add the below code in this file and save it.

eureka.instance.hostname=${vcap.application.uris[0]:localhost}

eureka.instance.nonSecurePort=80

eureka.instance.metadataMap.instanceId=${vcap.application.instance\_id:${spring.application.name}:${spring.application.instance\_id:${server.port}}}

eureka.instance.leaseRenewalIntervalInSeconds = 5

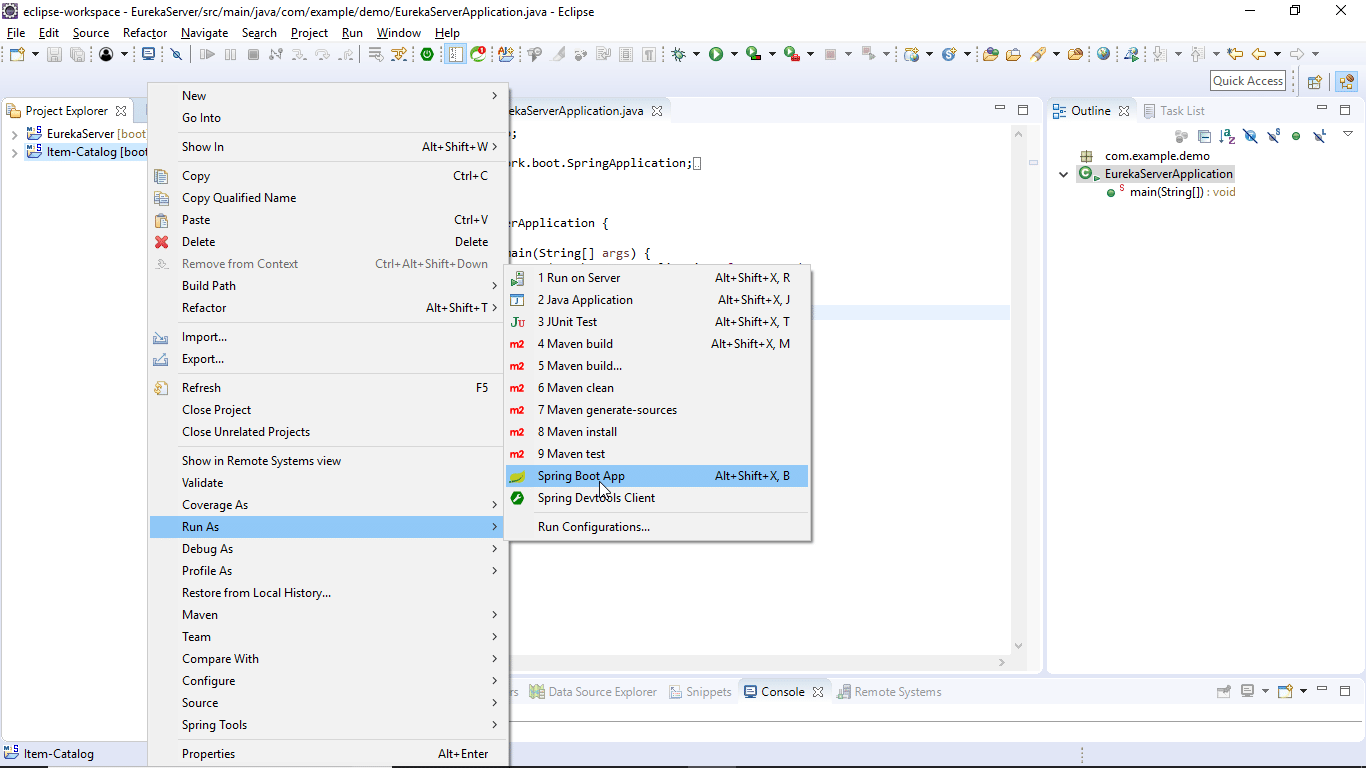
eureka.client.region = default

eureka.client.registryFetchIntervalSeconds = 5

eureka.client.serviceUrl.defaultZone=${vcap.services.pwa-eureka-service.credentials.uri}/eureka/

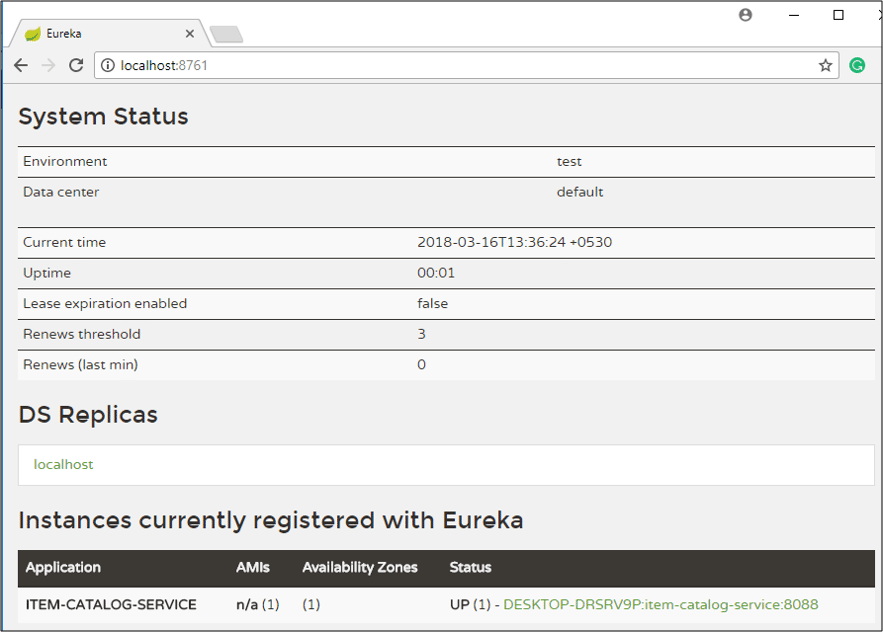
Now, to start the Application:

Right Click on Project -> **Run As** -> Click on " **Spring Boot App** "

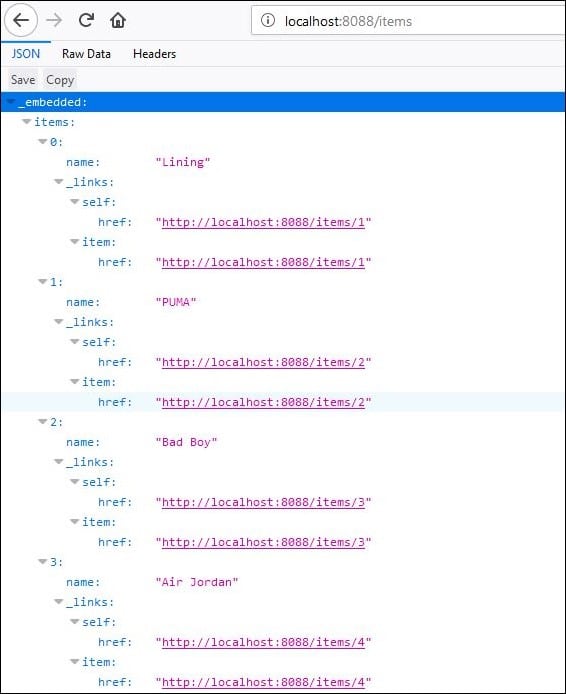


Note: In case of error try this step: Right Click on the **Project** -> **Run As** -> Click on "**Maven Build.**"

Now open http://localhost:8761. Here you will see Item Catalog service will be running.

****

You will see the list of items from the catalog service.

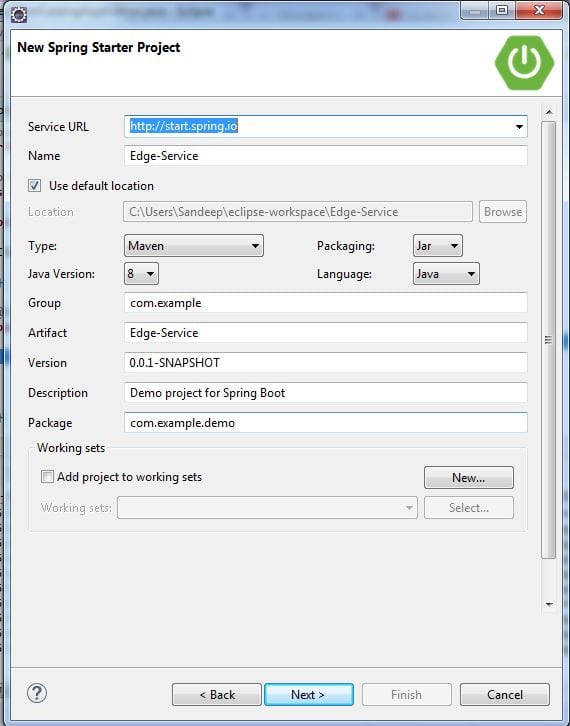


Now let us move forward and create the Edge Service.

**Creating an Edge Service**

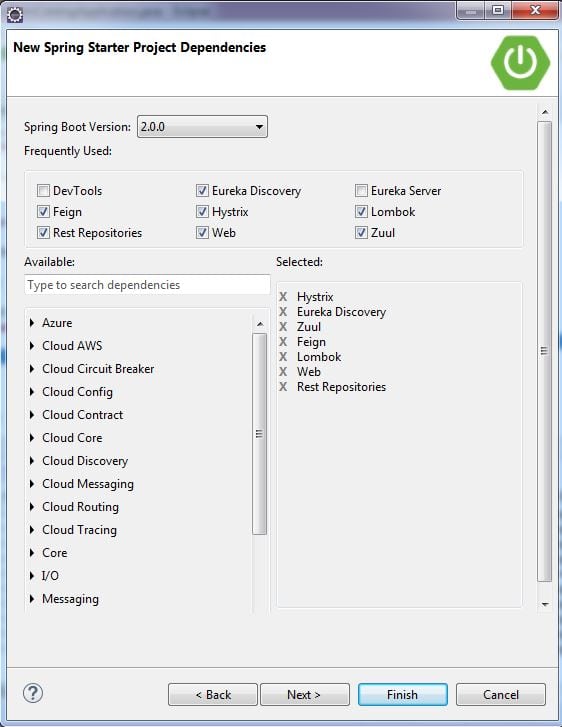
It is similar to the standalone Item service created in **Bootiful Development with Spring Boot and Angular**. However, it will have fallback capabilities which prevent the client from receiving an HTTP error when the service is not available.

Again create a new project. Use edge-service for the artifact name:



* **Eureka Discovery**: for service registration
* **Feign**: a declarative web service client
* **Zuul**: provides intelligent routing
* **Rest Repositories**: to expose JPA repositories as REST endpoints
* **Web**: Spring MVC and embedded Tomcat
* **Hystrix**: a circuit breaker to stop cascading failure and enable resilience
* **Lombok**: to reduce boilerplate code

Click on **Finish.**



Since the item-catalog-service is running on port 8088, you'll need to configure this application to run on a different port. Modify edge-service/src/main/resources/application.properties to set the port to 8089 and set an application name.

Now, Create the Cloud Properties file.

Click on**File -> New -> Other -> File**and add below code in this file and save it.

eureka.instance.hostname=${vcap.application.uris[0]:localhost}

eureka.instance.nonSecurePort=80

eureka.instance.metadataMap.instanceId=${vcap.application.instance\_id:${spring.application.name}:${spring.application.instance\_id:${server.port}}}

eureka.instance.leaseRenewalIntervalInSeconds = 5

eureka.client.region = default

eureka.client.registryFetchIntervalSeconds = 5

eureka.client.serviceUrl.defaultZone=${vcap.services.pwa-eureka-service.credentials.uri}/eureka/

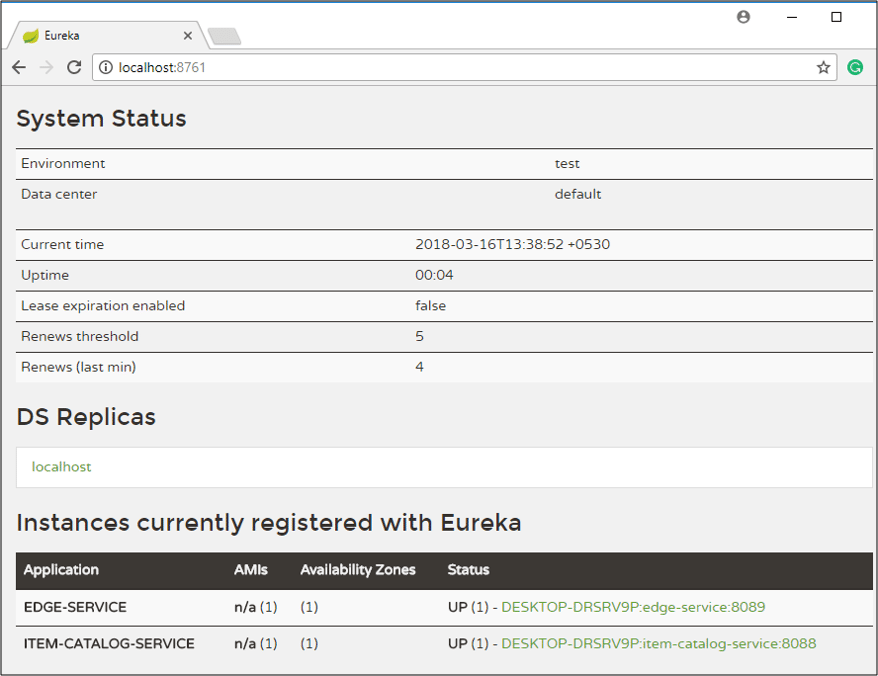
To enable Feign, Hystrix, and registration with the Eureka server, add the appropriate annotations to EdgeServiceApplication.java:

Create a Item DTO (Data Transfer Object) in this same file. Lombok's will generate a methods, getters, setters, and appropriate constructors.

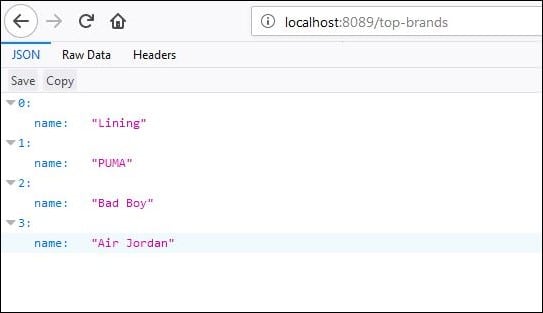
Create a ItemClient interface that uses Feign to communicate to the Item-catalog-service.

Create a RestController below the ItemClient that will filter out less-than-top brands and shows a /top-brands endpoint.

Start the edge-service application with Maven or your IDE and verify it registers successfully with the Eureka server.



Now invoke localhost:8089/top-brands, you will see the list of top brands from the catalog service.



**Note:**If you shut down the item-catalog-service application, you'll get a 500 internal server error.

To fix this, you can use Hystrix to create a fallback method and tell the goodItems() method to use it.

Restart the edge-service and you should see an empty list returned.

Start the item-catalog-service again and this list should eventually return the full list of top brands names.

Microservices implementation example with Spring Boot

Posted by: [Abhimanyu Prasad](https://www.javacodegeeks.com/author/abhimanyu-prasad) in [Software Development](https://www.javacodegeeks.com/category/software-development) December 29th, 2017 [1 Comment](https://www.javacodegeeks.com/2017/12/microservices-implementation-example-spring-boot.html#comments) 6112 Views

1. Introduction

We have already been through the [Microservice demo](https://www.javacodegeeks.com/2017/12/what-are-microservices.html) and it was so pleasing to see the wonderful response it received. Hopefully we are all good with the basics now. This tutorial will guide us all through the actual Microservices implementation, so we are well-directed every time we are asked to create it.

2. Understanding Service Registry

Traditionally, when we consume a REST service, we are usually provided with the network location of the service instance (the REST service URLs that are usually static). However, this tradition has changed with the Microservices architecture coming into picture. Lets understand, how!

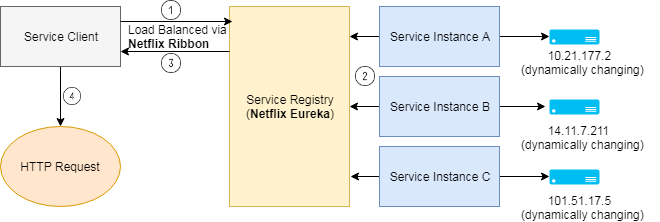
In Microservices architecture, network locations of the service instances are dynamically assigned/changed because of autoscaling, failures and upgrades. To comply with this dynamicity, **service discovery mechanism** comes in.

So there are two main service discovery patterns –

* client‑side discovery
* server‑side discovery

Client‑side discovery

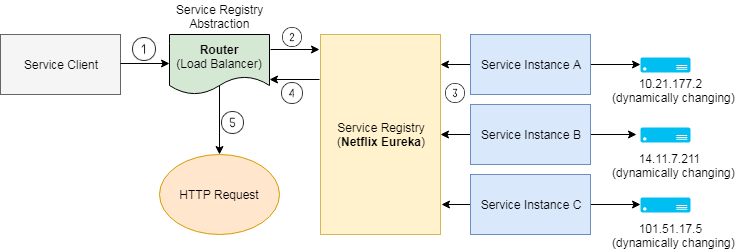
In client‑side service discovery, client queries the Service Registry and determine the available service instances using a separate load balancing mechanism. Load balancing usually works with Service Registry to load balance requests across the available service instances. Each HTTP request is done using one of the load balanced instance that is returned.



One of the major drawback is the tight coupling of client with Service Discovery and the Service discovery logic needs to be written at the client end.

Server‑side discovery

The client queries the Discovery Service via an abstraction layer, which is actually the load balancer or we can say a router, that queries the Service Registry and routes the HTTP request to an available service instance. Details of Service Discovery are abstracted (or hidden) from the client, as a result of which the client don’t need to write the discovery logic, which lead to more ideal loose coupling of client with the Service Discovery.



So what is Service Registry?

**Service registry** is the key part of service discovery mechanism. Service registry is a database of available service instances. Service registry assures that its highly available and up to date with the network locations of the services instances. Clients can cache network locations obtained from the service registry, but as the network locations keep changing, the cache data soon becomes out of date. To cope with this, it is the responsibility of Service Registry server (consists of a cluster of servers) to maintain the consistency and keep refreshing the network location of the service instances (usually refresh is done every 30 seconds).

One of most common example of a Service Registry is **Netflix Eureka**. It provides a REST API to register (using a POST request) and query (using an HTTP GET request) service instances.

Netflix achieves the high availability by running one or more Eureka servers in each [Amazon EC2 availability zone](http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html). Each Eureka in turn runs on an EC2 instance that has elastic IP address. DNS TEXT records are used to store the Eureka cluster configuration (maps to the list of network locations of Eureka servers), so that when Eureka server starts up, DNS is queried to retrieve the Eureka cluster configuration (network locations of Eureka servers) and assigns itself an unused elastic IP address.

**Hashicorp’s Consul** and **Apache Zookeeper** are other examples of Service Registry.

**Netflix Ribbon** is an IPC (Inter Process Communication) client that works with Eureka to load balance requests across the available service instances. The **@LoadBalanced**annotation configures the RestTemplate to use **Ribbon**, which has been configured to use the Eureka client to query service discovery and fetch available service instances.

3. Microservices Implementation

We will check out a simple demo on Microservices using Spring Boot. Hope we get more close to the Microservices concept.

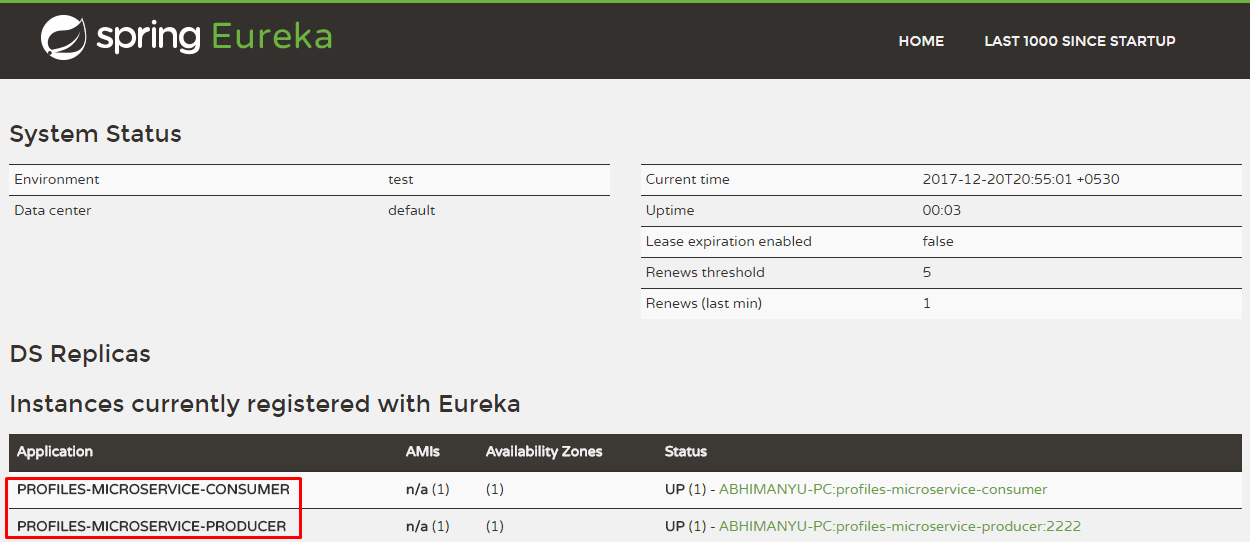
To create a Microservices system, we need to ensure of the below steps –

1. **Creation of Eureka Discovery Server**
   * With Spring Boot, just one annotation **@EnableEurekaServer** does the job.
2. **Creating Producer Microservice**
   * Register itself with Discovery Service
   * **@EnableDiscoveryClient** activates the Netflix Eureka DiscoveryClient implementation
3. **Create Consumer Microservice** which finds the producer service instance registered with Discovery Service
   * Register itself with Discovery Service
   * **@EnableDiscoveryClient** activates the Netflix Eureka DiscoveryClient implementation
   * Requests for DiscoveryClient instance of Producer Microservice using a smart **RestTemplate**.
4. We can then **test end-to-end result** by starting the Eureka service first. Once the Eureka service starts up, start the discovery clients one after the other.

4. Testing the application

Once the Discovery service and Discovery client are started, we can check if the Discovery clients got properly registered themselves with the Eureka Discovery server.

<http://localhost:1111/>



With Spring Boot all the implementation becomes so easy, just the right annotation at the right place with minimal manual configuration.

We will now try running the application as mentioned below –

<http://localhost:8080/>

Project Location: <https://github.com/abhi435/SpringBootMicroservicesDemo>

